DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

6 de diciembre de 2015 — Organizacion del computador II

Te voy a dar un Byte

Integrante	LU	Correo electrónico
Gonzalez Benitez, Albertito Juan	324/14	gonzalezjuan.ab@gmail.com
Lew, Axel Ariel	225/14	axel.lew@hotmail.com
Noli Villar, Juan Ignacio	174/14	juaninv@outlook.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

0.1. Introduccion

En este trabajo practico procederemos a realizar un sistema operativo que pueda correr un juego.

1. Ejercicio 1:

1.1. Introducción:

En este ejercicio vamos a realizar la Tabla de Descriptores Globales (*GDT*). Se pide que realizemos lo siguiente :

a) Que la tabla *GDT* tenga 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 500MB de memoria. Por último se pide no usar las primeras siete posiciones de la *GDT*, ya que se consideran utilizadas.

b) Pasar a modo protegido y setear la pila del kernel en la dirección 0x27000.

c) Agregar a la *GDT* un segmento adicional y escribir una rutina qe utilice este nuevo segmento para pintar la esquina superior izquierda de la pantalla.

d) Limpiar la pantalla y pintar el área del mapa (sugerido el color gris) junto con las barras inferiores para los jugadores.

1.2. Ítem a): Setear la *GDT*

Para este ítem completamos el archivo *GDT*.c proporcionado por la catedra. En el mismo la *GDT* es representada mediante un array de 30 posiciones. Cada posicion tiene la siguiente estructura.

```
[GDT_IDX_NULL_DESC] = (gdt_entry) {
    (unsigned short)   0x0000,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,       /* base[23:16]  */
    (unsigned char)    0x00,       /* type         */
    (unsigned char)    0x00,       /* s            */
    (unsigned char)    0x00,       /* dpl          */
    (unsigned char)    0x00,       /* p            */
    (unsigned char)    0x00,       /* limit[16:19] */
    (unsigned char)    0x00,       /* avl          */
    (unsigned char)    0x00,       /* l            */
    (unsigned char)    0x00,       /* db           */
    (unsigned char)    0x00,       /* g            */
    (unsigned char)    0x00,       /* base[31:24]  */
},
```

Figura 1: Este descriptor corresponde a la primer entrada de la *GDT*

Como la primer posicion de la tabla *GDT* debe ser corresponder a una entrada nula, llenamos la primer posicion como muestra la imagen debajo.

Luego, creamos los 4 segmentos que se piden a partir de la posicion 8 de la *GDT*, ya que por enunciado, no se deben tocar las primeras 7 posiciones de la table de descriptores. Mostramos en las imagenes de abajo como creamos un descriptor de datos y otro de codigos.

```
[GDT_IDX_NULL_DESC+8] = (gdt_entry) {
    (unsigned short)   0xF400,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,        /* base[23:16]  */
    (unsigned char)    0x0A,        /* type         */
    (unsigned char)    0x01,        /* s            */
    (unsigned char)    0x00,        /* dpl          */
    (unsigned char)    0x01,        /* p            */
    (unsigned char)    0x01,        /* limit[16:19] */
    (unsigned char)    0x00,        /* avl          */
    (unsigned char)    0x00,        /* l            */
    (unsigned char)    0x01,        /* db           */
    (unsigned char)    0x01,        /* g            */
    (unsigned char)    0x00,        /* base[31:24]  */
},
```

Figura 2: Este descriptor corresponde al segmento de datos de nivel 0

```
[GDT_IDX_NULL_DESC+9] = (gdt_entry) {
    (unsigned short)   0xF400,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,        /* base[23:16]  */
    (unsigned char)    0x02,        /* type         */
    (unsigned char)    0x01,        /* s            */
    (unsigned char)    0x00,        /* dpl          */
    (unsigned char)    0x01,        /* p            */
    (unsigned char)    0x01,        /* limit[16:19] */
    (unsigned char)    0x00,        /* avl          */
    (unsigned char)    0x00,        /* l            */
    (unsigned char)    0x01,        /* db           */
    (unsigned char)    0x01,        /* g            */
    (unsigned char)    0x00,        /* base[31:24]  */
},
```

Figura 3: Este descriptor corresponde al segmento de código de nivel 0

Los otros dos que faltan son exactamente iguales, solo que en la linea correspondiente al nivel (dpl) ponemos 0x03, ya que corresponde al nivel 3 de prioridad.

Los descriptores de segmentos tienen la siguiente forma:

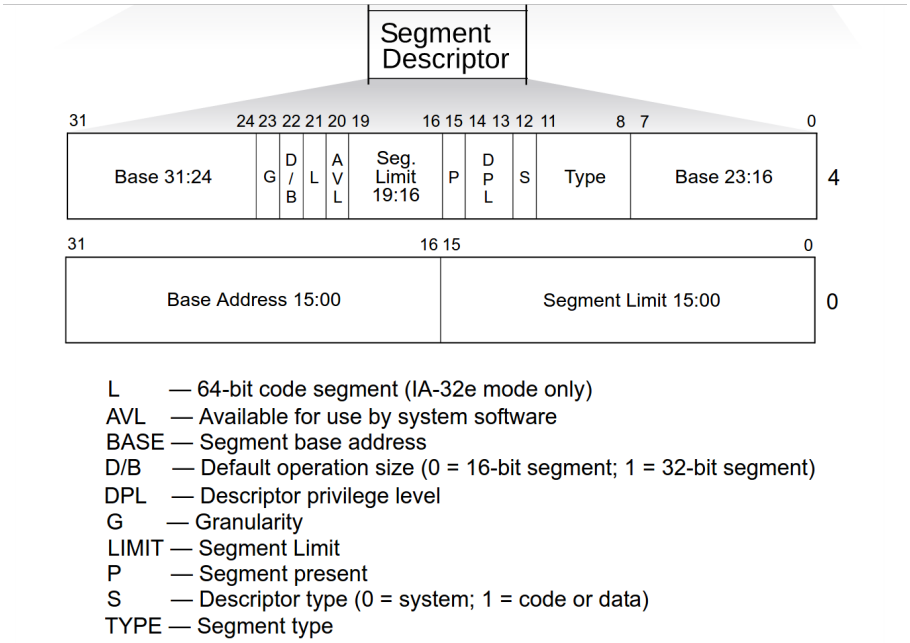

Figura 4: Este descriptor corresponde a la primer entrada de la *GDT*

Completamos nuestros descriptores como marcan las figuras 3 y 4 de esa forma porque:

Base: En el tp se pide que los descriptores direcciones los primeros 500mb de memoria. Por ende la base corresponde a la dirección 0x00000000.

G: Para poder direccionar 500mb, no nos alcanza la cantidad de bits que hay para el limite, por ende necesitamos activar la granularidad para poder abarcar más memoria, ya que cuando esta activada la posicion que indica el limite se multiplica por 4kb.

Límite: Como esta activada la granularidad, podemos abarcar 500mb de memoria, el limite correspondiente a 500mb con la granularidad activada es 0x0F400.

Type: Aqui se indica si el descriptor es de código/datos, a los correspondientes a datos les pusimos que eran de tipo 0x02 (segmento de datos de escritura/lectura) y a los de código que eran de tipo 0x0A (segmentdo de código de escritura/lectura).

S: Con este bit se decide si es un segmento de sistema (s=0) o si son de código/data (s=1), por ende a este bit le corresponde un 0x1.

Dpl: En esta seccion se declara el privilegio del segmento, a los que eran de nivel 0 les corresponde un 0x00 y a los de nivel 3 un 0x03

P: Este es el bit de present. Cuando es '1' el segmento correspondiente esta presente en la memoria RAM. Si es '0', el segmento esta en la memoria virtual. Por ende lo seteamos en 1.

Avl: Es el bit correspondiente a Available. Como no lo vamos a tener en cuenta lo dejamos en 0.

L: Indica si el código es de 64bits o de 32. Como trabajamos en 32bits dejamos este bit en 0.

D/B: Este bit define el tamaño de las operaciones en las que va a trabajar el procesador. De nuevo, como nos encontramos trabajando en 32bits, el tamaño de las operaciones debe ser de 32, por eso lo seteamos en 1.

1.3. Ítem b): Pasar a modo protegido y setear la pila

Para pasar a modo protegido realizamos los siguientes pasos:

a) Deshabilitamos las interrupciones, para eso utilizamos la instrucción CLI.

b) Cargamos el registro *GDTR* con la dirección base de la *GDT* utilizando la instrucción *LGDT* de esta manera:

```
lgdt [GDT DESC]
```

Donde GDT DESC es el descriptor de la tabla.

c) Seteamos el bit *PE* (BIT 0) del registro *CR0* en 1 para pasar a modo protegido. Procedemos a hacer esto con las intrucciones:

```
mov eax, cr0
or eax, 1
mov cr0, eax
```

d) Realizamos un far jump para cargar en el registro *CS* la dirección donde esta el segmento de código. Para esto utilizamos la instruccion:

```
jmp 0X40:modoprotegido
```

Donde 0x40 es la dirección donde en nuestra *GDT* compienza el segmento de código y modoprotegido es una etiqueta que se encuentra imediatamente debajo de este JMP.

e) Cargamos los registros de segmento de la siguiente manera.

```
mov ax, 0x48
mov ds, ax
mov ax, 0x48
mov ss, ax
```

Ahora que ya nos encontramos con el procesador trabajando en modo protegido, procedemos a setear la pila en la dirección 0x27000. Para eso seteamos los registros ebp y esp en la dirección 0x27000 con las siguientes instrucciones:

mov ebp, 0x27000
mov esp, 0x27000

1.4. Ítem c): Agregar a la *GDT* un segmento adicional y utilizarlo como memoria de vídeo

Para este ítem, agregamos a la *GDT* el siguiente segmento, el cual direcciona a la memoria de vídeo utilizada por la pantalla (Desde 0xB8000 a 0XC000).

```
// VIDEO
[GDT_IDX_NULL_DESC+12] = (gdt_entry) {
   (unsigned short)   0x8000,      /* limit[0:15]  */
   (unsigned short)   0x8000,      /* base[0:15]   */
   (unsigned char)    0x0B,       /* base[23:16]  */
   (unsigned char)    0x02,       /* type         */
   (unsigned char)    0x01,       /* s            */
   (unsigned char)    0x00,       /* dpl          */
   (unsigned char)    0x01,       /* p            */
   (unsigned char)    0x02,       /* limit[16:19] */
   (unsigned char)    0x00,       /* avl          */
   (unsigned char)    0x00,       /* l            */
   (unsigned char)    0x01,       /* db           */
   (unsigned char)    0x00,       /* g            */
   (unsigned char)    0x00,       /* base[31:24]  */
},
```

Figura 5:

Luego cargamos en un registro de segmento, en este caso *FS*, la posicion del segmento anterior. Por ultimo con las siguiente instrucción podemos poner en la esquina superior izquierda de la pantalla lo que querramos

mov word[fs:0x00], *aimprimir*

Donde *aimprimir* es algo de tamaño word, y su valor es el que aparece en la esquina superior izquierda de la pantalla. Esto ocurre porque se carga el segmento que comienza en la dirección a que la dirección 0xB8000 y se le suma el offset 0 y esta es la primer dirección de la memoria de vídeo utilizada por la pantalla.

Si por ejemplo escribimos la siguiente línea:

mov word[fs:0x00], 0xdb00

Obtenemos este resultado, que es lo que pedía el ejercicio.

Figura 6:

1.5. Ítem d): Limpiar la pantalla y pintar el área del mapa

En este punto creamos una función auxiliar en C para limpiar la pantalla. La función pinta la pantalla de gris y es la siguiente:

```
void aux_limpiarPantalla(){
  int i = 1;
  while (i<45){
    int j = 0;
    while(j<80){
      p[i][j].c = 219;
      p[i][j].a = 7;
      j++;
    }
    i++;
  }
```

Figura 7:

Esta función la escribimos en *screen.c* y utiliza la matriz P creada por la catedra para acceder a las posiciones de la pantalla. Estos son los resultados luego de utilizar nuestra función.

Figura 8:

Para pintar el mapa utilizamos la función dada por la catedra *screen_inicializar*. Para utilizar estas funciones, escribimos en *kernel.asm* las siguientes lineas con el siguiente resultado:

call aux_limpiarPantalla
call screen_inicializar

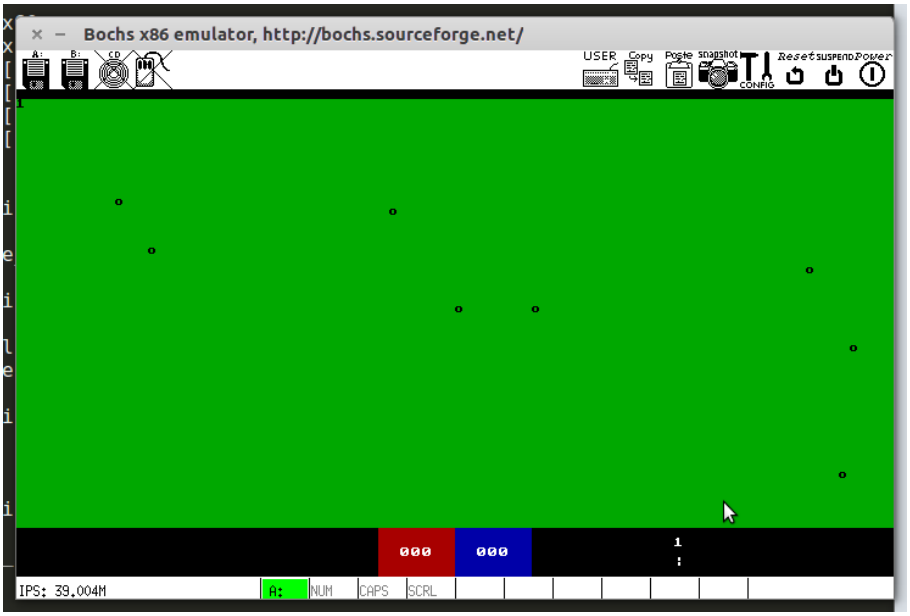

Figura 9:

2. Ejercicio 2:

2.1. Introducción:

En este ejercicio vamos a setear las interrupciones basicas, osea las reservadas para el procesador. Para esto completaremos la *IDT*. Se pide que realizemos lo siguiente :

a) Completar la *IDT* de tal forma que indique por pantalla el problema que se produjo y que interrumpa la ejecución.

b) Cargar la *IDT* y probarla.

2.2. Ítem a): Setear la *IDT*

Para esto utilizamos los siguientes archivos dados por la catedra *idt.c*, *isr.h*, *isr.asm* y los completamos.

2.2.1. *Idt.c:*

En este archivo se encuentra la función *IDT_ENTRY(número, dpl)* la cual recive el numero de la interrupción y su prioridad. La misma se encontraba incompleta y necesitaba ser completada con la información necesaria. La llenamos agregandole el segmento correspondiente y los atributos correpondientes. El segmento que le correpondia se ubicaba en la direccion 0x40 (explicado en el ejercicio 1) y los atributos son los correpondientes a el valor 0x8700, ya que el formato de los atributos de una interrupcion debe ser de la siguiente forma:

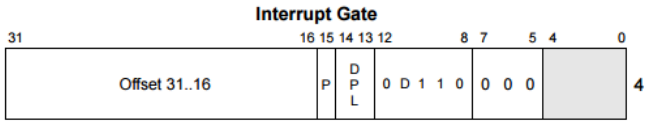

Figura 10: Formato de los atributos de una interrupcion

Entonces como 0x8E00 representa en binario 1000 1110 0000 0000, tiene el formato que queremos. Ya que P debe ser 1, DPL debe ser 00, luego sigue un 0, luego D (que en este caso es 1 ya que estamos trabajando en 32bits) luego siguen dos unos y por último nueve ceros, lo cual forma el número que queremos. De esta forma nos queda la funcion *IDT_ENTRY* de la siguiente manera:

```
#define IDT_ENTRY(numero, dpl)                                                  \
  idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);      \
  idt[numero].segsel = (unsigned short) 0x40;                                                                \
  idt[numero].attr = (unsigned short) 0x8E00 | (((unsigned short)(dpl & 0x3)) << 13);                        \
  idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF);
```

Figura 11: Formato de los atributos de una interrupcion

Por último, creamos las 20 interrupciones correspondientes utilizando *IDT_ENTRY* .

```
void idt_inicializar() {
    // Excepciones
    IDT_ENTRY(0, 0);
    IDT_ENTRY(1, 0);
    IDT_ENTRY(2, 0);
    IDT_ENTRY(3, 0);
    IDT_ENTRY(4, 0);
    IDT_ENTRY(5, 0);
    IDT_ENTRY(6, 0);
    IDT_ENTRY(7, 0);
    IDT_ENTRY(8, 0);
    IDT_ENTRY(9, 0);
    IDT_ENTRY(10, 0);
    IDT_ENTRY(11, 0);
    IDT_ENTRY(12, 0);
    IDT_ENTRY(13, 0);
    IDT_ENTRY(14, 0);
    IDT_ENTRY(15, 0);
    IDT_ENTRY(16, 0);
    IDT_ENTRY(17, 0);
    IDT_ENTRY(18, 0);
    IDT_ENTRY(19, 0);
```

Figura 12: Formato de los atributos de una interrupcion

2.2.2. *Isr.h:*

En este archivo tuvimos que declarar las siguientes funciones:

```
#ifndef __ISR_H__
#define __ISR_H__

void _isr0();
void _isr1();
void _isr2();
void _isr3();
void _isr4();
void _isr5();
void _isr6();
void _isr7();
void _isr8();
void _isr9();
void _isr10();
void _isr11();
void _isr12();
void _isr13();
void _isr14();
void _isr15();
void _isr16();
void _isr17();
void _isr18();
void _isr19();

#endif /* !__ISR_H__ */
```

Figura 13: Formato de los atributos de una interrupcion

Sino, la función del archivo anterior, *IDT_ENTRY*, no compilaba, pues no las encontraba. No es necesario que estas funciones hagan algo, ya que en realidad *IDT_ENTRY* las utilizaba como macro.

2.2.3. *Isr.asm:*

Por último, en este archivo, atendemos las interrupciones con sus correspondientes rutinas, como en el enunciado solo se pide que se interrumpa la ejecucion del programa y se muestre por pantalla la iterrupcion que genero el problema, solamente utilizamos una funcion macro dada por la catedra que simplemente muestra dicho mensaje y ejecuta un loop infinito.

La macro es la siguiente:

```
exception1 db    'Divide Error', 0


%macro ISR 1
global _isr%1

_isr%1:
   mov eax, %1
   push 0xf
   push 0
   push 0
   push exception%1
   call print
   sub esp, 4
   sub esp, 4
   sub esp, 4
   sub esp, 4
   jmp $

%endmacro
```

Figura 14: Macro para atender interrupciones

La macro de la imágen anterior reemplaza %1 por el primer parametro que recibe. Este parametro es el número correpsondiente a la interrupción, por ende cuando llegue la interrupucion "x", se pushea el mensaje exception"x" y print muestra por pantalla el mensaje correspondiente a la etiqueta exception"x", para luego quedarse saltando infinitamente y por lo tanto interrupiendo la ejecucion del programa. En la imagen se muestra un ejemplo del mensaje que se mostraria si se produjera la interrupcion 1 , correspondiente a la division por cero.

Por ultimo escribimos la rutina de atencion de interrupciones, la cual simplemente llama a la macro anterior, tomando como parámetro el número de la interrupcion.

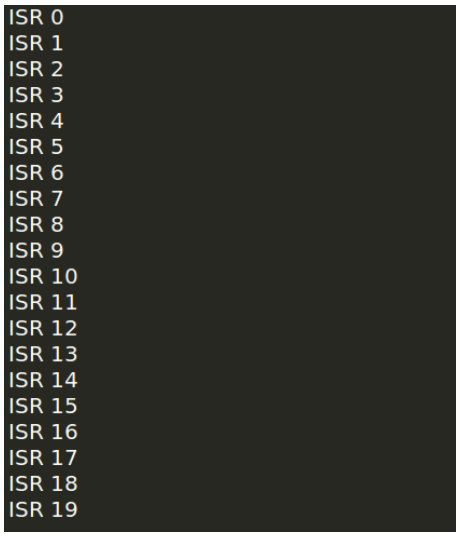

Figura 15: Rutina de atencion de interrrupciones.

2.3. Ítem b): Cargar y probar *IDT*

Para cargar la *IDT* y probarla, agregamos las siguientes lineas al *kernel*:

```
lidt [IDT_DESC]
mov eax, 0
mov ecx, 1
div ecx
```

Obteniendo el resultado esperado:

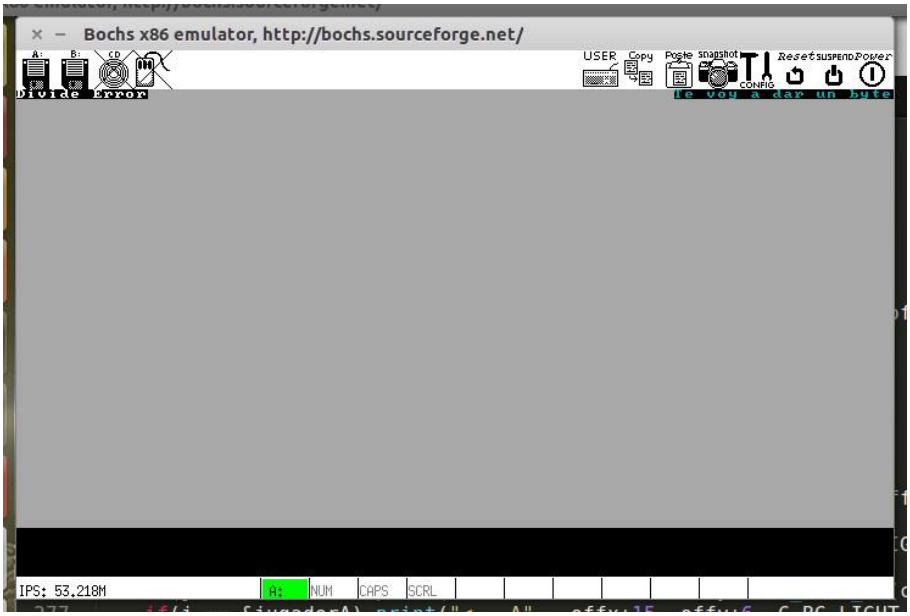

Figura 16: Rutina de atencion de interrrupciones.

3. Ejercicio 3:

3.1. Introducción:

En el siguiente ejercicio activaremos la páginacion básica. Completaremos el archivo *mmu.c* para relizar los siguientes ítems.

a) Limpiar el buffer de video y pintar el mapa.

b) Completar la función *mmu_mapear_página(unsigned int virtual, unsigned int cr3, unsigned int física)*

c) Utilizando la función anterior, completar la función *mmu_inicializar_dir_kernel* generar un directorio de páginas que mapee, usando identity mapping, las direcciones 0x00000000 a 0x003FFFFF. El directorio de páginas a inicializar se encuentra en la dirección 0x27000.

d) Activar la paginación y verificar que el sistema sigue funcionando imprimiendo el nombre del grupo.

e) Completar la función *mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)*

f) Probar la función anterior desmapeando la última página del kernel (0x3FF000).

3.2. Ítem a): Limpiar el buffer de video y pintar el mapa *IDT*

Este ítem ya lo implementamos anteriormente, el mapa ya se encuentra pintado. Lo hicimos en el ejercicio 2, al utilizar las funciones *aux_limpiarPantalla* y *screen_inicializar*.

3.3. Ítem b): Completar la función *mmu_mapear_página(unsigned int virtual, unsigned int cr3, unsigned int física)*

En este punto realizamos la siguiente función:

```
void mmu_mapear_pagina(uint virtual, uint cr3, uint fisica, uint attrs){
        uint *pagDir = (uint *) ((cr3 & 0xFFFFF000) + ((virtual >> 22 )*4));

        uint *pageTable;
        uint *pageTableEntry;
        if ( *pagDir % 2 == 1){


                pageTableEntry = (uint *) ((*pagDir & 0xFFFFF000) + ((virtual >> 12) & 0x000003FF)*4);
                uint pageDirAux = *pagDir;
                pageDirAux = pageDirAux & 0xFFFFFFFE;
                uint atr_aux = attrs;
                atr_aux = atr_aux >> 1;
                if (pageDirAux % 4 == 0 && atr_aux % 2 == 1){                   // si r/w == 0 y attrs es de r/w
                        *pagDir = *pagDir | 10;

                }
                atr_aux = atr_aux >> 1;
                pageDirAux = pageDirAux & 0xFFFFFFFC;
                if (pageDirAux % 8 != 0 && atr_aux % 2 == 0){                   // si u/s == 1 y attrs es de supervisor
                        *pagDir = *pagDir & 0xFFFFFFFB;
                }
        } else {
                pageTable = (uint *) mmu_proxima_pagina_fisica_libre();
                *pagDir = ((uint )pageTable & 0xFFFFF000) | 0x00000007;

                pageTableEntry = pageTable + ((virtual >> 12) & 0x000003FF)*4;
                mmu_inicializar_pagina(pageTable);
        }



        *pageTableEntry = 0x00000000 | fisica;
        *pageTableEntry = (*pageTableEntry & 0xFFFFF000) | attrs;

}
```

Figura 17: Función mapeadora

Lo que realiza esta función es lo siguiente:

A: Recibe como parámetro una dirección virtual, una física, un *CR3* y la informacion sobre los atributos.

B: Como *CR3* tiene la siguiente forma:

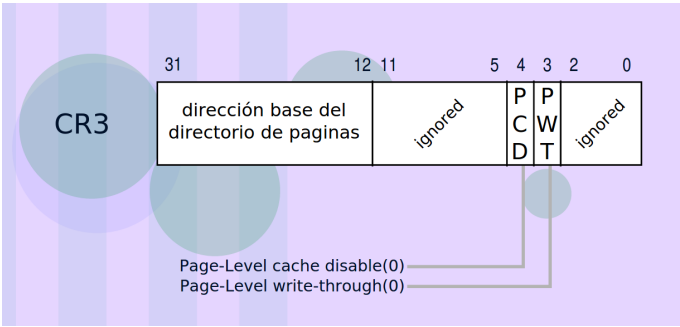

Figura 18: Formato *CR3*

Le hacemos un and lógico con 0xFFFFF000, de esta forma nos quedamos solo con la direccion base del directorio de páginas del cr3. Luego shifteamos la direccion virtual del parámetro de entrada en 22 posiciones, para quedarnos con el offset de la misma y multiplicamos este

offset por 4 pues los punteros de las páginas de directorio ocupan 32 bytes y cada posicion de memoria es de 8.

Ahora sumamos la direccion base del directorio de páginas con el offset anterior y obtenemos la dirección de la tabla de página que buscabamos.

```
uint *pagDir = (uint *) ((cr3 & 0xFFFFF000) + ((virtual >> 22 )*4));
```

Figura 19: Formato *CR3*

C: Lo siguiente es fijarse si la direcion de la tabla de página obtenida existe o no. Para esto nos fijamos que en modulo 2 la direccion anterioirmente obtenida. Asi obtenemos el útlimo bit de la misma, si es 1 (P) sabemos que esta presente y si es 0 sabemos que no lo esta (¬P) y tenemos que crearla

P: Si esta presente, tenemos que encontrar la posición de la tabla de página deseada. Para eso, realizamos un and lógico con la dirección obtenida anteriormente y 0xFFFFF000 para obtener la direccion base de la tabla de página (en la cual buscaremos la posicion deseada).

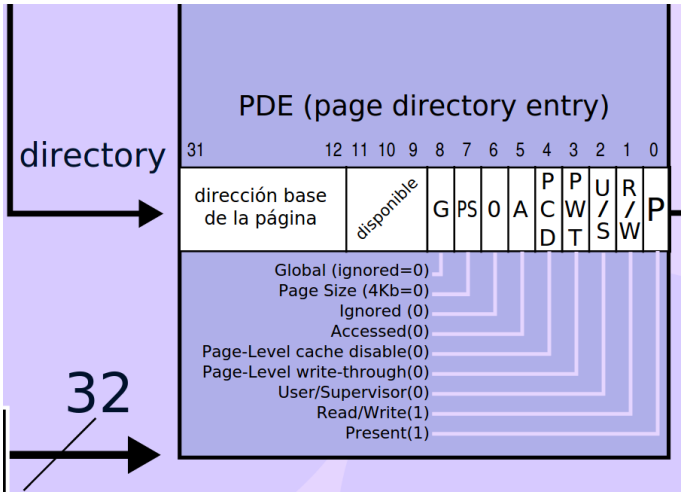

Figura 20: Formato de una posicion del directorio de tabla de páginas

Luego shiftemos 12 posiciones la direccion vitual del parámetro de entrada para obtener el segundo offset necesesario y le hacemos un and lógico con 0x000003FF ara que solo quede la informacion que queremos y limpiar la posible basura que haya.

Este offset lo multiplicamos por 4 por la misma razón explicada anteriormente y se lo sumamos a la direccion base obtenida antes. Asi obtenemos la posicion de la tabla de página deseada.

En este caso, es necesario checkear si los atributos que se encuentran en la posicion recientemente encontrada son los mismos que los que se encontraban en la posicion del directorio de páginas encontrada en B. Obtenemos los atributos, y los comparamos, y en caso de ser necesario los cambiamos.

```
uint pageDirAux = *pagDir;
pageDirAux = pageDirAux & 0xFFFFFFFE;
uint atr_aux = attrs;
atr_aux = atr_aux >> 1;
if (pageDirAux % 4 == 0 && atr_aux % 2 == 1){
        *pagDir = *pagDir | 10;

}
atr_aux = atr_aux >> 1;
pageDirAux = pageDirAux & 0xFFFFFFFC;
if (pageDirAux % 8 != 0 && atr_aux % 2 == 0){
        *pagDir = *pagDir & 0xFFFFFFFB;
}
```

Figura 21: Comparacion y cambio de atributos

¬P: Si la página no estaba presente, en necesario crearla. Entonces pedimos una dirección para crearla con la función *mmu_proxima_página_fisica_libre()*. Luego ponemos esta direccion obtenida en el directorio de tabla de páginas, en la posicion obtenida en B, con los atributos correspondientes (para eso el or lógico con 0x00000007)

```
uint mmu_proxima_pagina_fisica_libre(){
        pagLibre += 4096;
        cantPagLibre--;
        return pagLibre-4096;
}
```

Figura 22: Función que nos da una posicion libre para poner una página. La variable pagLibre es una variable global inicializada con el valor 0x100000, y cantPaglibre en 768

Por último obtenemos la posicion deseada de la tabla de página obtenida en B de la misma manera que cuando la página esta presente e incicializamos la tabla de página llenandola de ceros.

```
void mmu_inicializar_pagina(uint * pagina){
        int i = 0;
        while(i<4096){
                *pagina = 0x00000000;
                pagina += 32;
                i++;
        }
}
```

Figura 23: Función que inicializa una página

D: Ahora que tenemos todo en orden y la posición de la tabla de página deseada, procedemos a poner en esta posición la direccion física pasada por parámetro con los atributos tambien pasados por parámetro.

```
*pageTableEntry = 0x00000000 | fisica;
*pageTableEntry = (*pageTableEntry & 0xFFFFF000) | attrs;
```

Figura 24: Identity mapping

Como es Identity Mapping alcanza con poner en esta posición el resultado de hacer un or lógico con la direccion física y luego un or lógico con los atributos.

3.4. Ítem c): Inicializar el directorio de páginas en 0x00027000 y mapear las direcciones desde la 0 hasta la 0x003FFFFF con Identity Mapping*IDT*

Para realizar lo pedido completamos la funcion *mmu_inicializar_dir_kernel* mostrada a continuación:

```
uint mmu_inicializar_dir_kernel(){

        mmu_inicializar_pagina((uint *)0x00027000);
        uint cr3 = 0x00027000;
        uint attrs = 0x007;
        int i = 0x00000000;
        while (i<1024){
                mmu_mapear_pagina(i*4096,cr3,i*4096,attrs);
                i++;
        }

        paginaJugadorA = mmu_proxima_pagina_fisica_libre();
        paginaJugadorB = mmu_proxima_pagina_fisica_libre();
        return cr3;

}
```

Figura 25:

Esta funcion comienza creando una tabla de páginas vacía en la dirección 0x27000, la cual va a actuar como directorio de páginas. Luego crea la variable *Attrs* y la inicializa con el valor 0x007, el cual corresponde a los atributos que queremos que tenga el directorio de páginas, osea que P = 1, sea de lectura y escritura y sea de sistema. Tambien crea la variable *Cr3* el cual se le asigna el valor 0x00027000 para que tenga la direccion base y los atributos correspondientes al directorio de paginas que queremos inicializar.

Para mapear cada posición del directorio recientemente creado se llama a la función del ítem anterior . LLamamos a esta función con las variables *Cr3* y *Attrs* anteriormente creadas, y en cada iteracion se le da una posicion nueva del directorio para que haga el correpsondiente mapeo.

Por último es necesario que cada jugador tenga una página, por eso le asignamos a cada uno una página libre.

3.5. Ítem d): Activar la paginación y verificar que el sistema sigue funcionando imprimiendo el nombre del grupo.

Activamos la paginacion escribiendo en el kernel las siguientes líneas:

Inicializar el directorio de paginas:

call mmu_inicializar_dir_kernel

Cargar directorio de paginas:

```
mov eax, 0x00027000
mov cr3, eax
```

Habilitar paginacion:

```
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Luego verificamos que el sistema sigue funcionando imprimiendo por pantalla el nombre del grupo

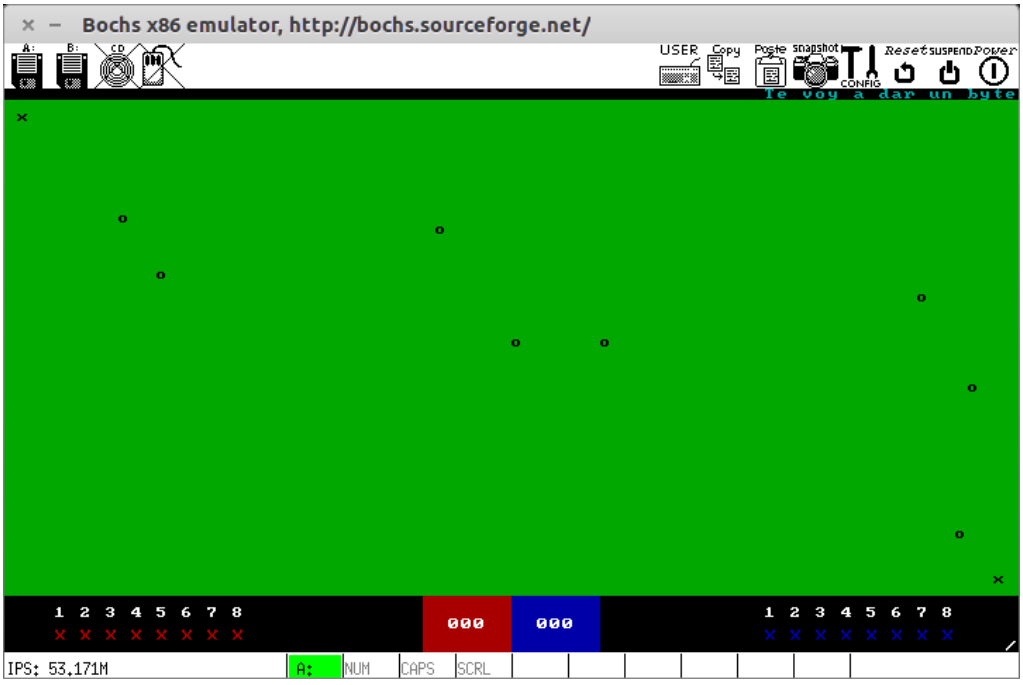

Figura 26:

3.6. Ítem e): Completar la función *mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)*

Para unmapear una dirección virtual realizamos la siguiente función

```
uint mmu_unmapear_pagina(uint virtual, uint cr3){
        uint *pagDir = (uint *) ((cr3 & 0xFFFFF000) + ((virtual >> 22 )*4));

        uint *pageTableEntry;
        if ( *pagDir % 2 == 1){                                 // ESTA PRESENTE?
                pageTableEntry = (uint *) ((*pagDir & 0xFFFFF000) + ((virtual >> 12) & 0x000003FF)*4);
        } else {
                return 0;
        }
        *pageTableEntry = 0x00000000;
        return 0;
}
```

Figura 27: Identity mapping

La cual realiza el mismo cálculo que en el ítem 2 para acceder a la posicion de tabla de pagina buscada en el directorio de páginas y luego si esta pagina esta presente se hace un calculo tambien explicado anteriormente para acceder a la posicion deseada en esta pagina. Una vez obtenida esta posicion se procede a llenarla con ceros.

3.7. Ítem f): Probar la función anterior desmapeando la última página del kernel (0x3FF000).

Para probar la función anterior, escribimos en el kernel las siguientes lineas:

Le pasamos los parametros a la funcion:

```
mov eax, cr3
push eax
mov eax, 0x3FF000
push eax
```

La llamamos:

```
call mmu_unmapear_pagina
```

Limpiamos pila:

```
pop eax
pop eax
```

De esta manera llamamos a la función, y para ver que los resultados son los deseados, usamos el comando *Info tab* de *bochs* para ver hasta donde llega el mapeo de paginas y obetemos que llega hasta la direccion 0x3FEFFF en vez de llegar hasta 0x3FEFFF, por lo tanto podemos decir que la función hizo lo devido.

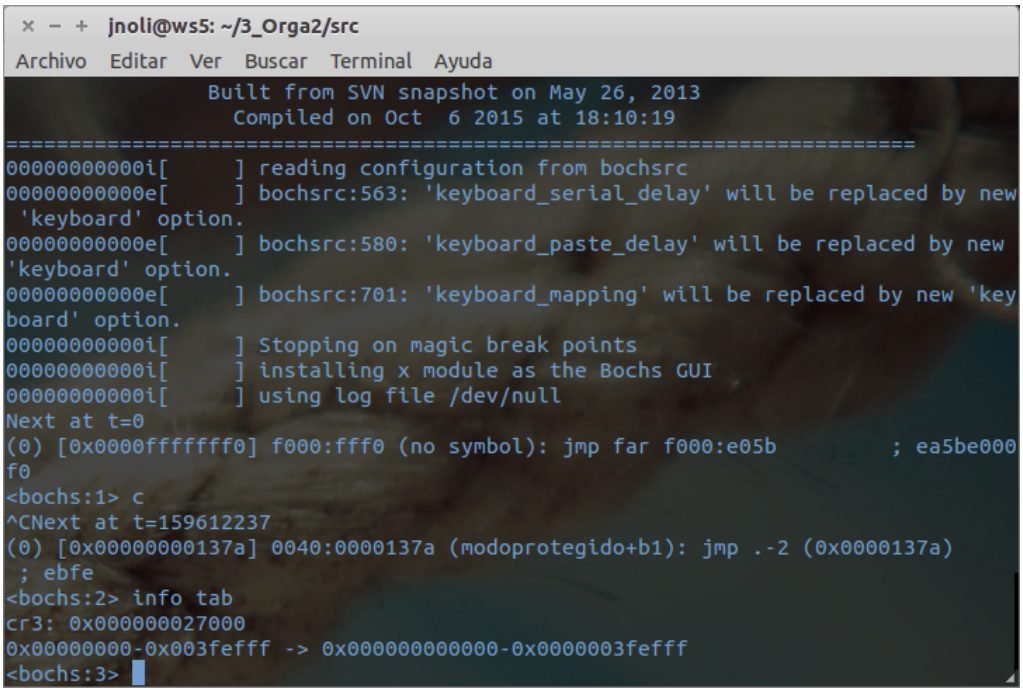

Figura 28: Identity mapping

4. Ejercicio 4:

4.1. Introducción:

a) Completar *inicializar_mmu* que se encargue de inicializar las estructuras globales necesarias para administrar la memoria en el área libre (un contador de páginas libres).

b) Completar la función *mmu_inicializar_memoria_perro*.

4.2. Ítem a): Completar *inicializar_mmu*.

Para realizar esta función simplemente creamos en el archivo *mmu.c* las variables globales las cuales puden ser usadas por el resto de las funciones:

uint pagLibre = 0x100000
uint cantPagLibre = 768
uint paginaJugadorA
uint paginaJugadorB

4.3. Ítem b): Completar la función *mmu_inicializar_memoria_perro*.

La función la realizamos de la siguiente manera:

```
uint mmu_inicializar_memoria_perro(perro_t *perro, int index_jugador, int index_tipo){
        uint *pagDir = (uint *) mmu_proxima_pagina_fisica_libre();     // PIDO UNA PAGINA LIBRE
        mmu_inicializar_pagina(pagDir);         // LIMPIO PAGINA
        int i = 0x00000000;
        while (i<1024){                                 // HAGO IDENTITY MAPPING
                mmu_mapear_pagina(i*4096,(uint )pagDir,i*4096,0x007);
                i++;
        }

        uint aCopiar;
        if (index_jugador == 1){
                if (perro->tipo == 1){                  // TAREA A1
                        aCopiar = 0x10000;
                } else {                                                // TAREA A2
                        aCopiar = 0x11000;
                }
                mmu_mapear_pagina(0x400000, (uint) pagDir, paginaJugadorA, 0x007);
        } else{
                if (perro->tipo == 1){                  // TAREA B1
                        aCopiar = 0x12000;
                } else {                                                // TAREA B2
                        aCopiar = 0x13000;
                }
                mmu_mapear_pagina(0x400000, (uint) pagDir, paginaJugadorB, 0x007);
        }
        uint dondeCopiar = 0x500000 + (perro->jugador->x_cucha + perro->jugador->y_cucha*80)*4;

        int j = 0;
        while (j<3520){                                 // MAPEO CON EL MAPA
                mmu_mapear_pagina(0x800000+j*4096,(uint )pagDir,j*4096+0x500000,0x007);
                j++;
        }

        mmu_mapear_pagina(0x401000, (uint) pagDir, dondeCopiar, 0x007);
        mmu_copiar_pagina(aCopiar,0x401000);
        return (uint ) pagDir;
}
```

Figura 29: Función principal

La función realiza lo siguiente:

A: Recibe como parametros un perro, un int que sirve para indicar que jugador es y otro int para identificar el tipo de perro.

B: Comienza pidiendo un lugar para crear una página libre, para eso llama a la función *mmu_proxima_pagina_fisica_l* Una vez obtenida la posicion, procede a crear la pagina y luego la mapea con la funcion *mmu_mapear_pagina*

```
uint *pagDir = (uint *) mmu_proxima_pagina_fisica_libre();
mmu_inicializar_pagina(pagDir);         // LIMPIO PAGINA
int i = 0x00000000;
while (i<1024){                                  // HAGO IDENTITY MAPPING
        mmu_mapear_pagina(i*4096,(uint )pagDir,i*4096,0x007);
        i++;
}
```

Figura 30:

C: Luego, para saber que tarea copiar identifica que jugador es, usando el parámetro de entrada *index_jugador*, luego identifica que tipo de perro es , usando la variable de entrada *index_tipo*. Con esa informacion, la función sabe donde esta la tarea que tiene que copiar. Ademá es necesario que se le mapee esta nueva tarea al jugador correspondiente por eso, dependiendo de que jugador sea, se le mapea esta nueva tarea.

```
uint aCopiar;
if (index_jugador == 1){
        if (perro->tipo == 1){                  // TAREA A1
                aCopiar = 0x10000;
        } else {                                                // TAREA A2
                aCopiar = 0x11000;
        }
        mmu_mapear_pagina(0x400000, (uint) pagDir, paginaJugadorA, 0x007);

} else{
        if (perro->tipo == 1){                  // TAREA B1
                aCopiar = 0x12000;
        } else {                                                // TAREA B2
                aCopiar = 0x13000;
        }
        mmu_mapear_pagina(0x400000, (uint) pagDir, paginaJugadorB, 0x007);

}
```

Figura 31:

D: Después de mapear el mapa devuelta es necesario mapear la dirección 0X401000 con la posición donde tiene que estar la tarea. Para saber en que dirección tiene que estar la tarea (o sea dónde va a estar el perro), la función toma la posición 0x500000 (posición donde se encuentra el mapa) y le suma el valor X y el valor Y de la cucha del perro y mapea 0X401000 con la dirección obtenida. De esta manera el kernel sabe donde deben estar los perros y finalmente se puede copiar la tarea a la dirección 0X401000, la cual va a estar mapeada correctamente al lugar donde hay que poner a los nuevos perros/tareas.

5. Ejercicio 5:

5.1. Introducción:

En este ejercicio nos encargaremos del manejo de las interrupcíones del reloj, teclado y otra interrupcíon de software 0x46. Se pide:

a) Completar las entradas necesarias en la *IDT*.

b) Escribir la rutina asociada a la interrupcíon de reloj, de manera que por cada tick, se muestre la animacion de un cursor rotando

c) Escribir la rutina asociada a la interrupcíon de teclado, para aquellas teclas a utilizar en el juego, para que se imprima la misma en la pantalla

d) Escribir la rutina asociada a la interrupcíon de software 0x46 para que modifique el valor de *eax* por 0x42

5.2. Ítem a): Completar la *IDT*

Comenzamos agregando las interrupcíones a la *IDT*. Utilizamos para el reloj y el teclado las posiciones 32 y 33 respectivamente, pues son las primeras disponibles no utilizadas por el procesador. Para la interrupcíon por software, utilizamos la posicion 70, pues esta se llamara al llamar a int 0x46 = 70. Las declaramos con privilegio 0, pues son de sistema y no se querria que algo externo al sistema los controle.
Para esto utilizamos la macro anteriormente explicada, con los siguientes parametros:

```
IDT_ENTRY(32, 0);
IDT_ENTRY(33, 0);
IDT_ENTRY(70, 0);
```

Además, las definimos en *isr.h*, para luego escribir su rutina.

```
void _isr32();
void _isr33();
void _isr70();
```

5.3. Ítem b): Rutina del reloj

El código es el siguiente:

```
global _isr32
_isr32:

    pushad
    call fin_intr_pic1
    call game_atender_tick
    call screen_actualizar_reloj_global
    popad
    iret
```

Cada vez que se llame, la interrupcíon hara lo siguiente. Pusheara todos los registros de proposito general (lo cual en realidad no es necesario, dado que la interrupcíon no los modifica). Llamara a la fnción *fin_intr_pic1* para decir al pic que la interrupcíon fue atendida. Luego llamara a las fnciónes *game_atender_tick* y *screen_actualizar_reloj_global* para mostrar la animacion por pantalla. Luego popeara los registros y volvera de la interrupcíon con la instruccion iret

5.4. Ítem c): Rutina del teclado

El código es el siguiente:

```
global _isr33
_isr33:
    pushad
    call fin_intr_pic1
    in al, 0x60
    push eax
    call imprim
    add esp, 4
    popad
    iret
```

Igual que la interrupcíon anterior, comienza pusheando todos los registros. En este caso, es necesario preservar *eax*, pues lo modificamos. Nuevamente llamamos a *fin_intr_pic1*. Luego utilizamos la operacion *in*, para mover al registro al el Leemos del teclado a traves del puerto 0x60 y obtenemos el scan code en *eax*. Pusheamos este valor y llamamos a la función *imprim* creada por nosotros. Esta fnción se encarga de traducir el scan code en una letra (en caso de que sea válido) y luego llama a la función print para mostrarlo por pantalla. Finalmente restauramos la pila, popeamos los registros y volvemos de la interrupcíon.

5.5. Ítem d): Rutina 0x46

El código es el siguiente:

```
global _isr70
_isr70:
    mov eax, 0x46
    iret
```

Lo único que hace esta interrupción es modificar el registro *eax*. Por lo tanto no es necesario salvar los registros. Esta interrupción sera modificada mas adelante

6. Ejercicio 6:

6.1. Introducción:

En este ejercicio trabajaremos con todo lo relacionado a la *TSS*, realizaremos los siguientes ítems:

a) Definir las entradas en la *GDT* que considere necesarias para ser usadas como descriptores de *TSS*.

b) Completar la entrada de la *TSS* de la tarea Idle con la información de la tarea Idle. La tarea Idle se encuentra en la dirección 0x00016000. La pila se alojará en la misma dirección que la pila del kernel y debe compartir el mismo CR3 que el kernel.

c) Construir una función que complete una TSS libre con los datos correspondientes. El código de las tareas se encuentra a partir de la dirección 0x00010000. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base de la tarea. Para el mapa de memoria se debe construir uno nuevo utilizando la función *mmu_inicializar_dir_perro*. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0.

d) Completar la entrada de la *GDT* correspondiente a la tarea_inicial.

e) Completar la entrada de la *GDT* correspondiente a la tarea Idle.

f) Escribir el código necesario para ejecutar la tarea Idle, es decir, saltar intercambiando las TSS, entre la tarea_inicial y la tarea Idle.

g) Modificar la rutina de la interrupción 0x46, para que implemente los servicios según se indica en la sección 4.4, sin desalojar a la tarea que realiza el syscall.

h) Ejecutar una tarea perro manualmente. Es decir, crearla y saltar a la entrada en la *GDT* de su respectiva TSS.

6.2. Ítem a): Definir entradas de la *GDT* necesarias

Para este punto definimos 18 nuevas entradas en la *GDT*.

Las primeras dos nuevas entradas de la *GDT* corresponden a la *Tarea inicial* y a la tarea *idle* respectivamente.

```
// TSS DESCRIPTOR TAREA_INICIAL (B = 0, DPL = 0, AVL = 0, G = 0)
[GDT_IDX_NULL_DESC+12] = (gdt_entry) {
    (unsigned short)    0x0067,         /* limit[0:15]  */
    (unsigned short)    0x0000,         /* base[0:15]   */
    (unsigned char)     0x00,           /* base[23:16]  */
    (unsigned char)     0x9,            /* type         */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* dpl          */
    (unsigned char)     0x01,           /* p            */
    (unsigned char)     0x00,           /* limit[16:19] */
    (unsigned char)     0x00,           /* avl          */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* g            */
    (unsigned char)     0x00,           /* base[31:24]  */
},

// TSS DESCRIPTOR IDLE (B = 0, DPL = 0, AVL = 0, G = 0)
[GDT_IDX_NULL_DESC+13] = (gdt_entry) {
    (unsigned short)    0x0067,         /* limit[0:15]  */
    (unsigned short)    0x0000,         /* base[0:15]   */
    (unsigned char)     0x00,           /* base[23:16]  */
    (unsigned char)     0x9,            /* type         */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* dpl          */
    (unsigned char)     0x01,           /* p            */
    (unsigned char)     0x00,           /* limit[16:19] */
    (unsigned char)     0x00,           /* avl          */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* g            */
    (unsigned char)     0x00,           /* base[31:24]  */
},
```

Figura 32: *Entrada de la GDT correspondiente a la tarea inicial y a la idle*

Las siguientes 8 entradas corresponden a los 8 perros del **jugador A** ordenadas en orden decreciente en el número de perros, es decir, el *id* y finalmente las últimas 8 entradas corresponden a los 8 perros del **jugador B** también ordenadas decrecientemente por *id*.

```
// TSS DESCRIPTOR PERRO A 8 (B = 0, DPL = 0, AVL = 0, G = 0)
[GDT_IDX_NULL_DESC+14] = (gdt_entry) {
    (unsigned short)    0x0067,         /* limit[0:15]  */
    (unsigned short)    0x0000,         /* base[0:15]   */
    (unsigned char)     0x00,           /* base[23:16]  */
    (unsigned char)     0x9,            /* type         */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* dpl          */
    (unsigned char)     0x01,           /* p            */
    (unsigned char)     0x00,           /* limit[16:19] */
    (unsigned char)     0x00,           /* avl          */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x0,            /*              */
    (unsigned char)     0x00,           /* g            */
    (unsigned char)     0x00,           /* base[31:24]  */
},
```

Figura 33: *Entrada de la GDT correspondiente al descriptor de tss de la tarea perro*

6.3. Ítem b): Completar la *TSS* de la tarea *idle* y de la *tareainicial*

Para este item hacemos lo siguiente, primero completamos el campo *base* de la entrada de la *GDT* correspondiente a la tarea *idle*.Para ello le asignamos la dirección 0x00016000 tal como

indica el enunciado.
Como también nos dicen que comparte el *esp* con el *kernel* entonces en la entrada *esp* de la *TSS* le asignamos 0x27000 que era el *esp* asignado al kernel.Además como comparten el *cr3* le asigno a la entrada correspondiente en la *TSS* 0x28000.
La siguiente imagen muestra como queda completo el descriptor de la *TSS*

```
void tss_inicializar() {

    // modificamos aca la gdt
    // tarea inicial
    gdt[12].base_0_15 = (uint )&tss_inicial & 0x0000FFFF;
    gdt[12].base_23_16 = ((uint )&tss_inicial & 0x00FF0000) >> 16;
    gdt[12].base_31_24 = ((uint )&tss_inicial & 0xFF000000) >> 24;
    // idle
    gdt[13].base_0_15 = (uint )&tss_idle & 0x0000FFFF;
    gdt[13].base_23_16 = ((uint )&tss_idle & 0x00FF0000) >> 16;
    gdt[13].base_31_24 = ((uint )&tss_idle & 0xFF000000) >> 24;

    tss_idle.esp = 0x27000;
    tss_idle.ebp = 0x27000;
    tss_idle.cr3 = 0x27000;
    tss_idle.eip = 0x16000;
    tss_idle.esp0 = 0x27000;
    tss_idle.ds = 0x48;
    tss_idle.ss0 = 0x48;
    tss_idle.ss = 0x48;
    tss_idle.fs = 0x48;
    tss_idle.gs = 0x48;
    tss_idle.es = 0x48;
    tss_idle.cs = 0x40;
    tss_idle.eflags = 0x202;
```

Figura 34: *Completamos el desscriptor de laTSS correspondiente a la tarea iddle y a la inicial*

6.4. Ítem c): Realizar la función *tss_completar(int jugador, int perro, perro_t* perro)*

Lo que nos piden en este punto es hacer la función *void tss_completar(int jugador, int perro, perro_t* perro)*
Lo primero que hacemos en esta función es pedir una nueva página libre para usarla como una pila de nivel 0 para la tarea.
Lo siguiente es preguntar si es la tarea (el perro) es una tarea correspondiente al **jugador A** o al **jugador B**. Para saber donde guardar el descriptor. Para cada perro de ambos jugadores se realiza lo siguiente:

- En los campos *cs*, *es*, *gs*, ss, *ds*, *fs* tienen los segmentos definidos anteriormente en la *GDT*.
- La entrada *esp* tiene 0x0402000-12.
- La entrada *eip* tiene 0x00401000.
- La entrada *eflags* = 0x202.
- La entrada *esp0* = Es la posicion de la página libre pedida mas 4kb pues tiene apilado en la pila los argumentos.
- La entrada *iomap* = 0xFFFF.
- La entrada *ss0* = 0x48.

- El nuevo *cr3* va a ser el devuelto por la función *mmu_inicializar_memoria_perro*. Asignamos el *cr3* al campo correspondiente y actualizamos la entrada correspondiente a la *GDT* para esa tarea seteando los campos base e índice.

La siguiente imágen muestra dicho proceso.

```
void tss_completar(int jugador, int perro, perro_t *rrope){
    uint espCero = mmu_proxima_pagina_fisica_libre();
    int posicion = perro;
    if (jugador == 0 ){
        tss_jugadorA[posicion].cs = 0x5B;
        tss_jugadorA[posicion].es = 0x53;
        tss_jugadorA[posicion].gs = 0x53;
        tss_jugadorA[posicion].ss = 0x53;
        tss_jugadorA[posicion].ds = 0x53;
        tss_jugadorA[posicion].fs = 0x53;
        tss_jugadorA[posicion].eax = 0x0;
        tss_jugadorA[posicion].ebx = 0x0;
        tss_jugadorA[posicion].ecx = 0x0;
        tss_jugadorA[posicion].edx = 0x0;
        tss_jugadorA[posicion].esi = 0x0;
        tss_jugadorA[posicion].edi = 0x0;
        tss_jugadorA[posicion].esp = 0x0402000-0xC;
        tss_jugadorA[posicion].eip = 0x00401000;
        tss_jugadorA[posicion].eflags = 0x202;
        tss_jugadorA[posicion].esp0 = (espCero+4096);
        tss_jugadorA[posicion].iomap = 0xFFFF;
        tss_jugadorA[posicion].ldt = 0x00000000;
        tss_jugadorA[posicion].ss0 = 0x48;

        uint nuevoCr3 = mmu_inicializar_memoria_perro(rrope, jugador, perro);
        tss_jugadorA[posicion].cr3 = nuevoCr3;

        gdt[rrope->id].base_0_15 =   (uint )&tss_jugadorA[posicion] & 0x0000FFFF;
        gdt[rrope->id].base_23_16 = ((uint )&tss_jugadorA[posicion] & 0x00FF0000) >> 16;
        gdt[rrope->id].base_31_24 = ((uint )&tss_jugadorA[posicion] & 0xFF000000) >> 24;
```

Figura 35:

6.5. Ítem d): Completar la entrada de la *tarea_inicial* en la *GDT*

La entrada de la *GDT* correspondiente a la tarea inicial en principio la definimos con cualquier valor en la base pero con los valores correspondientes en los demás campos.

```
// TSS DESCRIPTOR TAREA_INICIAL (B = 0, DPL = 0, AVL = 0, G = 0)
[GDT_IDX_NULL_DESC+12] = (gdt_entry) {
    (unsigned short)    0x0067,      /* limit[0:15]  */
    (unsigned short)    0x0000,      /* base[0:15]   */
    (unsigned char)     0x00,       /* base[23:16]  */
    (unsigned char)     0x9,        /* type         */
    (unsigned char)     0x0,        /*              */
    (unsigned char)     0x00,       /* dpl          */
    (unsigned char)     0x01,       /* p            */
    (unsigned char)     0x00,       /* limit[16:19] */
    (unsigned char)     0x00,       /* avl          */
    (unsigned char)     0x0,        /*              */
    (unsigned char)     0x0,        /*              */
    (unsigned char)     0x00,       /* g            */
    (unsigned char)     0x00,       /* base[31:24]  */
},
```

Figura 36: *Entrada de la GDT correspondiente a la tarea inicial*

Luego con la función *tss_inicializar()* le asignamos la base correspondiente.

```
// tarea inicial
gdt[12].base_0_15 = (uint )&tss_inicial & 0x0000FFFF;
gdt[12].base_23_16 = ((uint )&tss_inicial & 0x00FF0000) >> 16;
gdt[12].base_31_24 = ((uint )&tss_inicial & 0xFF000000) >> 24;
```

Figura 37: *Entrada de la GDT correspondiente a la tarea inicial*

6.6. Ítem e): Completar la entrada de la *idle* en la *GDT*

Esta sección es similar a la anterior. Creamos una entrada en la *GDT* para esta tarea y en la función *tss_inicializar()* seteamos los valores correspondientes para su base en la *GDT* y también seteamos su TSS.
Como nos dicen que la tarea se encuentra en la dirección 0x00010000 ese es el valor que ponemos en la base y por el enunciado va a compartir el CR3. Lo mismo con su pila.

```
// idle
gdt[13].base_0_15 = (uint )&tss_idle & 0x0000FFFF;
gdt[13].base_23_16 = ((uint )&tss_idle & 0x00FF0000) >> 16;
gdt[13].base_31_24 = ((uint )&tss_idle & 0xFF000000) >> 24;

tss_idle.esp = 0x27000;
tss_idle.ebp = 0x27000;
tss_idle.cr3 = 0x27000;
tss_idle.eip = 0x16000;
tss_idle.esp0 = 0x27000;
tss_idle.ds = 0x48;
tss_idle.ss0 = 0x48;
tss_idle.ss = 0x48;
tss_idle.fs = 0x48;
tss_idle.gs = 0x48;
tss_idle.es = 0x48;
tss_idle.cs = 0x40;
tss_idle.eflags = 0x202;
```

Figura 38: *Seteo la TSS de la tarea idle con los valores correspondientes*

6.7. Ítem f): Saltar a la tarea Idle

Esto es simplemente hacer un jump far en el archivo kernel.asm. Como el intercambio en los valores de la TSS lo hace automáticamente el procesador simplemente agregamos al archivo *kernel.asm* la siguiente linea

$$jmp\ 0x68{:}0$$

6.8. Ítem g): Completar la iterrución 0x46 de acuerdo al punto 4.4 del enunciado

Para este punto lo que hacemos en la interrupción 0x46 es pushear en la pila los parametros de la interrupcion que llegan en los registros EAX y ECX y llamar a la función $game\_syscall\_manejar(uintsyscall, uintpar$

Lo que hace la función es, dependiendo de los parámetros que le llega, llamar a la función correspondiente para atender la interrupción las cuales pueden ser:

- $game\_perro\_mover(tareaactual, parametro1delainterrupcion)$.
- $game\_perro\_cavar(tareaactual)$.
- $game\_perro\_olfatear(tareaactual)$.
- $game\_perro\_recibirorden(tareaactual)$.

El código es el siguiente:

```
uint game_syscall_manejar(uint syscall, uint param1)
{

if(syscall == 1){
        game_perro_mover(sched_tarea_actual(), param1);
}else if(syscall == 2){
        game_perro_cavar(sched_tarea_actual());
}else if(syscall == 3){
        game_perro_olfatear(sched_tarea_actual());
}else if(syscall == 4){

}
   return 0;
}
```

Figura 39: *Manejo de interrupciones*

Notar que la imagen anterior corresponde al código final, en el cual la interrupcion, como el Tp lo pide, salta a la tarea *idle* ,luego de haber atendido la interrupción, haciendo jmp 0x68:0. Cómo en este punto se pedía que la tarea actual no fuera desalojada, simplemente ignorar el jmp 0x68:0

6.9. Ítem h): Probar correr un perro

Este itém lo testeamos y todo anduvo bien, como no va a estar en el tp final borramos el código que lo implementaba, pero fue testeado.

7. Ejercicio 7:

7.1. Introducción:

En este ejercicio realizaremos el *scheduler*.

a) Construir una función para inicializar las estructuras de datos del *scheduler*.

b) Crear la función *sched_proxima_a_ejecutar()* que devuelve el índice de la próxima tarea a ser ejecutada.

c) Crear una función *sched_atender_tick()* que llame a *game_atender_tick()* pasando el número de tarea actual y luego devuelva el índice en la gdt al cual se deberá saltar. Reemplazar el llamado a *game_atender_tick* por uno a *sched_atender_tick* en el handler de la interrupción de reloj.

d) Modificar la rutina de la interrupción 0x46, para que implemente los servicios según se indica en la sección 4.4.13

e) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. El intercambio se realizará a según indique la función *sched_proxima_a_ejecutar().*

f) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y corran la próxima.

g) Implementar el mecanismo de debugging explicado en la sección 4.8 que indicará en pantalla la razón del desalojo de una tarea.

7.2. Ítem a): Inicializar el *scheduler*

Para incializar el scheduler completamos la funicón *void sched_inicializar()* del archivo sched.d. En el tp el scheduler es una estructura que posee un array de tareas denimoinado *tasks* y un int denominado *current* en el cual se guarda la tarea actual. Las tareas tambien son estructuras, y contienen un int para recordar la posicion en la que se guarda en la gdt y un puntero a la tarea que representan. Además le agregamos a esta estructura 3 ints, los cuales vamos a utilizar para saber lo siguiente: último jugador, último perro usado por el jugador A y último perro utilizado por el jugador B Esta estructura se puede obserbar en la siguiente imagen:

```
typedef struct sched_task_t
{
    unsigned int gdt_index;
  perro_t *perro;
} sched_task_t;

// el scheduler posee un arreglo de tareas (cada una puede estar libre o asignada)
typedef struct sched_t
{
    sched_task_t tasks[MAX_CANT_TAREAS_VIVAS+1];
    ushort current;
    ushort ultimoperroA;
    ushort ultimoperroB;
    ushort ultimojugador;
} sched_t;
```

Figura 40: Estructura del scheduler usado

La función en cuestión es la siguiente:

```
void sched_inicializar()
{
    scheduler.current = 0;
    scheduler.tasks[0].gdt_index = 13;
    scheduler.tasks[0].perro = NULL;
    scheduler.ultimojugador = 0;
    scheduler.ultimoperroA = 0;
    scheduler.ultimoperroB = 0;

    int i = 1;
    while(i<=MAX_CANT_TAREAS_VIVAS){
        scheduler.tasks[i].gdt_index = 13 + i;
        scheduler.tasks[i].perro = NULL;
        i++;
    }
}
```

Figura 41: *void sched_inicializar()*

Para inicializar la estructura del scheduler realiza lo siguiente:

A Setea el valor de *current* en 0, pues la tarea inicial debe ser la tarea *IDLE* y por convención decidimos que esta se encuentre en la posición 0 del array de tareas. Devido a la decisión anterior, setiamos en el vector de tareas que el gdt_index de la tarea 0 sea 13 (posición en la *GDT* de la tarea *IDLE*) y que el puntero a la tarea de la tarea 0 sea *NULL*.

B Setea los 3 ints que agregamos en 0 ya que nadie jugo todavía.

C Luego itera por todo el array de tareas *tasks* seteandole a cada tarea una posición en la gdt correspondiente y seteando el puntero decada una en *NULL*.

7.3. Ítem b): Crear la función *sched_proxima_a_ejecutar()*

Esta función comienza guardando cual es el jugador de la tarea actual y busca las siguiente tarea que corresponda al siguiente jugador. Para eso se fija cual fue el último perro usado por el jugador contrario e itera por todos los perros de este a partir del último utilizado. Si no llega a encontrar ningun perro empieza a iterar por los perros del jugador actual, tambien empezando a iterar desde el último perro utilizado por el jugador actual. Notar que para lograr esta forma de cambiar las tareas necesitabamos agregar los 3 ints que aclaramos al principio.

7.4. Ítem c): Crear una función *sched_atender_tick()*

Esta función debería devolver la posicion de la *GDT* de la proxima tarea, para eso utiliza la funcion del ítem anterior, además actualiza el int *current* de la estructura del sheduler y actualiza cual fue el último jugador y su último perro utilizado.

```
ushort sched_atender_tick()
{

   if(sched_tarea_actual() == scheduler.tasks[sched_proxima_a_ejecutar()].perro){
         return scheduler.tasks[scheduler.current].gdt_index;
   }else{
         scheduler.current = sched_proxima_a_ejecutar();
         return scheduler.tasks[scheduler.current].gdt_index;
   }

}
```

Figura 42: *Funcion proxima tarea a ejecutar*

7.5. Ítem d): Atender la interrupcion 0x46

Explicado en el ejercicio 6, ítem h, es igual solo que ahora no se ignora el jmp a la tarea *idle*

7.6. Ítem e): Realiza el intercabio e tareas

El intercambio de tareas lo realizamos en cada interrupción del reloj, utilizando las funciones explicadas anteriormente. Para utilizarlas escribimos las siguientes lineas en la *RAE* del reloj:

pushad
call fin_intr_pic1
call sched_tarea_actual
push eax
call game_atender_tick
call sched_atender_tick
shl ax, 3
str cx
cmp ax,cx
je .fin
mov [sched_tarea_selector], ax
jmp far:[sched_tarea_offset]
.fin:
call screen_actualizar_reloj_global
pop eax
popad
iret

Esta rutina pregunta por la poscicion de la *GDT* de la proxima tarea a ejecutar y si la proxima tarea a ejecutar es la misma que se esta ejecutando ahora no hace nada, sin embargo si es diferente realiza un *jmp far* utilizando como selector la posicion en la *GDT* correspondiente. De esta forma se guarda la *TSS* actual y se carga la nueva.

A esta rutina luego le agregamos más código para ver si el juego estaba frenado porque el modo debug había detectado que un perro habia generado una falta. No lo mostramos en este punto para que quede mas claro, además el modo debug no influye en este ítem.

7.7. Ítem f): Realiza el mecanismo de debug

Para este ítem realizamos varias cosas:

A : Creamos dos variables, $juegoFrenado$ y $modo_debug$, la primera vale 0 por defecto y pasa a valer 1 cuando el modo debug esta activado y se detecta que un perro causo una falta,

sirve para que si esta prendida, la rutina del reloj salte siempre a la tarea *idle*. La segunda simplemente indica si el modo debug esta activado o no, si esta activada el juego se pone en pausa y muestra la pantalla debug cuando un perro rompe el programa, sino, esta pantalla no se muestra y simplemente se elimina el perro.

B : Modificamos la *RAE* correspondiente a las primeras 20 interrupciones, para que de esta forma, en vez de hacer simplemente jmp $ salte a la funcion $set_frenado()$.

```
_isr%1:
    call setFrenado
    call deshabilitar_pic
    call resetear_pic
    call habilitar_pic
    sti
    jmp $


    iret
```

C : Creamos la funcion $set_frenado()$ la cual se fija si el modo debug esta activado, si lo esta, pone la variable $juegoFrenado$ en 1 e imprime la pantalla debug. Si el modo debug no esta activado simplemente elimina la tarea actual, la cual es la que rompió el juego. De esta manera el juego puede continuar.

```
void setFrenado(){
    if (sched_tarea_actual() == NULL){
        printf("%s\n", "Error");
        breakpoint();
    }
    if (modoDebug == 1){
        juegoFrenado = 1;
        pantallaDebug();
    } else {
        sched_remover_tarea(sched_buscar_gdt_tarea(sched_tarea_actual()));
    }

}
```

Figura 43: *Funcion proxima tarea a ejecutar*

D : Modificamos la rutina del reloj para que si la variable *juegoFrenado* esta en 1, salte directamente a la tarea *idle*, sin hacer nada mas.

```
global _isr32
_isr32:

    pushad
    call fin_intr_pic1

    call readFrenado
    cmp eax, 1
    je .estaFrenado

    call sched_tarea_actual
    push eax
    call game_atender_tick            ; llamo a atender tick con el pe
    call sched_atender_tick

    shl ax, 3
    str cx
    cmp ax,cx
    je .fin


    ;xchg bx, bx
    mov [sched_tarea_selector], ax
    jmp far [sched_tarea_offset]


    .fin:
       call screen_actualizar_reloj_global

    pop eax
    popad
        iret


    .estaFrenado:
        popad
        iret
```

E : Creamos la funcion *pantallaDebug*() la cual guarda la pantalla y luego imprime por pantalla el contenido de todos los registros y parte del stack de la tarea.

F : Realizamos la funcion *continuarJuego*() la cual se llama si se presiona la tecla *y* y el juego estaba frendo. Esta función setea *juegoFrenado* en 0, restaura la pantalla previamente guardada por *pantallaDebug* y elmina la tarea actual. Ahora como el juego ya no esta frenado, se pueden saltar a otras tareas que no sean la *idle*.

```
void continuarJuego(){
    juegoFrenado = 0;
    restaurarPantalla();
    sched_remover_tarea(sched_buscar_gdt_tarea(sched_tarea_actual()));
}
```

H : Modificamos la *RAE* del teclado, para que si el juego esta frenado solo se detecte la tecla y.

```
void imprim(char letra){
    if (juegoFrenado == 1){
        if (letra == 0x15){
            print("y",0,0,3);
            continuarJuego();
            return;
        } else {
            return;
        }
```